DCA0204, Módulo 1
Listas e Complexidade

Daniel Aloise

baseados em slides do prof. Leo Liberti, École Polytechnique, França

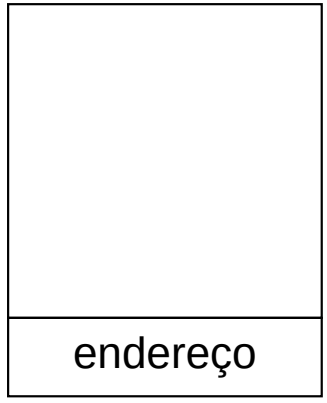
DCA, UFRN

Sumário

- Revisão
- Complexidade
- Listas

Revisão

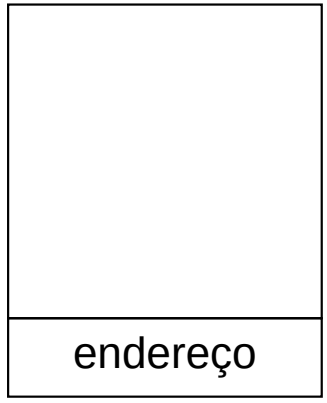
Memória



Célula de memória

- tem um endereço
- armazena um dado d

Memória

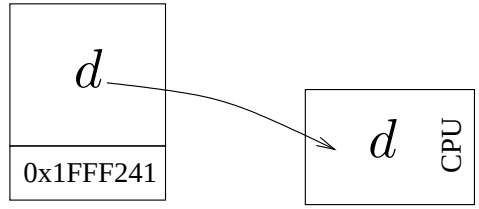


Célula de memória

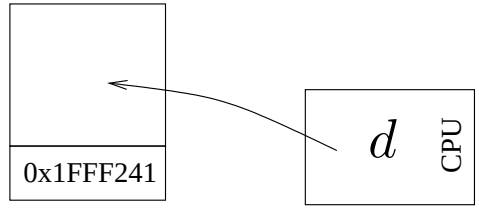
- tem um endereço
- armazena um dado d

Duas operações

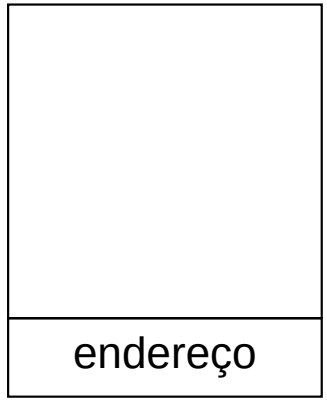
- Move dado da célula para a CPU (read)



- Move dado da CPU para a célula (write)



Memória

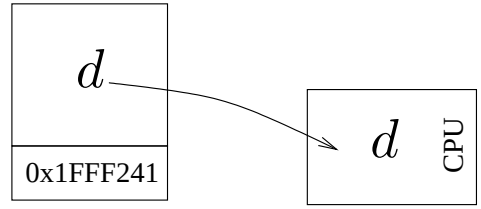


Célula de memória

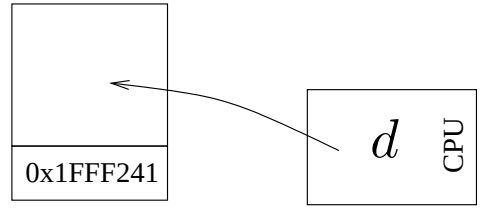
- tem um endereço
- armazena um dado d

Duas operações

- Move dado da célula para a CPU (read)



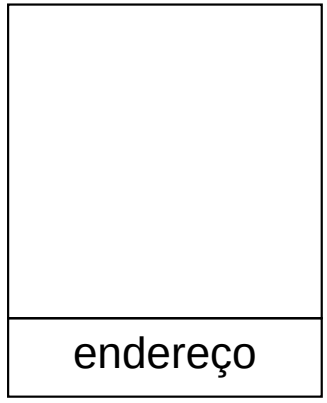
- Move dado da CPU para a célula (write)



Representação da memória: *uma sequência de células*

d_0	d_1	d_2	d_3	d_4	d_5
0x0	0x1	0x2	0x3	0x4	0x5

Memória

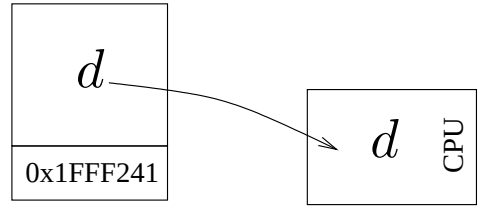


Célula de memória

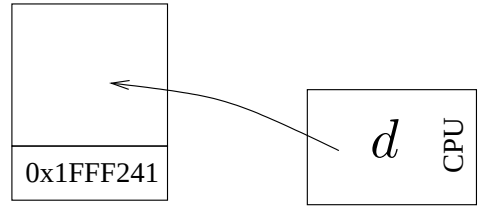
- tem um endereço
- armazena um dado d

Duas operações

- Move dado da célula para a CPU (read)



- Move dado da CPU para a célula (write)



Representação da memória: *uma sequência de células*

d_0	d_1	d_2	d_3	d_4	d_5
0x0	0x1	0x2	0x3	0x4	0x5

Uma função $D : \mathbb{A} \rightarrow \mathbb{D}$

$\mathbb{A}$: conjunto de endereços

$\mathbb{D}$: conjunto de dados

Hipóteses

- Iremos assumir que a memória do computador é infinita para propósitos teóricos.

→ Na prática ela é finita

- Cada dado pode ser armazenado em uma única célula

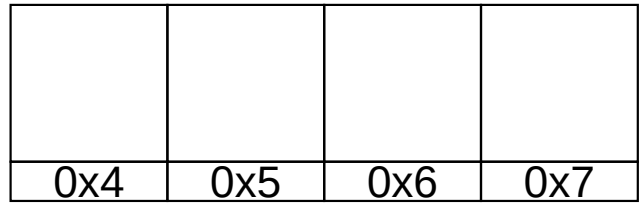
→

Diferentes elementos de dados podem ter tamanhos diferen

Nomeando memória

Uma variável de um programa é apenas um nome para um pedaço da memória

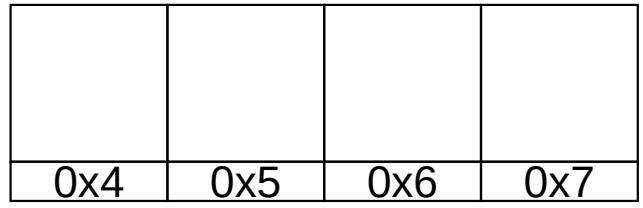
X denota:



Nomeando memória

Uma variável de um programa é apenas um nome para um pedaço da memória

X denota:

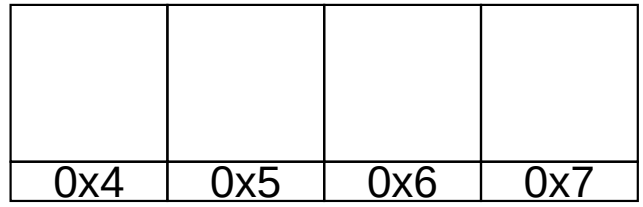


- Nós simplesmente associamos um nome ao endereço inicial
- O tamanho do pedaço de memória é dado pelo **type** do nome

Nomeando memória

Uma variável de um programa é apenas um nome para um pedaço da memória

X denota:



- Nós simplesmente associamos um nome ao endereço inicial
- O tamanho do pedaço de memória é dado pelo **type** do nome

- **Tipos básicos**: `int`, `long`, `char`, `float`, `double`
- **Tipos compostos**: Produtos cartesianos dos tipos básicos

if `y.a` $\in$ `int` and `y.b` $\in$ `float` then `y` $\in$ `int` $\times$ `float`

Operações Básicas

- **Atribuição**: escreve um valor na célula(s) de memória nomeadas pela variável (i.e. “variável=valor”)
- **Aritmética**: $+, -, \times, \div$ para números inteiros e de ponto flutuante
- **Teste**: avalia uma condição lógica: se verdadeiro, muda endereço para a próxima instrução a ser executada.
- **Salto do Loop**: ao invés de realizar a próxima instrução na memória, salta para a próxima instrução em um dado endereço

Operações Básicas

- **Atribuição**: escreve um valor na célula(s) de memória nomeadas pela variável (i.e. “variável=valor”)
- **Aritmética**: $+, -, \times, \div$ para números inteiros e de ponto flutuante
- **Teste**: avalia uma condição lógica: se verdadeiro, muda endereço para a próxima instrução a ser executada.
- **Salto do Loop**: ao invés de realizar a próxima instrução na memória, salta para a próxima instrução em um dado endereço

ATENÇÃO! *Nestes slides, “=” é usado para significas duas coisas diferentes*:

1. em atribuições, “variável $=$ valor” significa “por value na célula cujo endereço é nomeado por variável”
2. nos testes, “variável $=$ valor” é VERDADEIRO se a célula cujo endereço é nomeado por variável contem valor, and FALSO otherwise

em C/C++/Java “$=$” é usado para atribuições, e “$==$” para testes

Operações compostas: programas

Programas são construídos recursivamente a partir de operações básica

- Se A, B são operações, então concatenação “`A;B`” é uma operação

  Semântica: execute A, então execute B

Operações compostas: programas

Programas são construídos recursivamente a partir de operações básica

- Se A, B são operações, então concatenação "`A;B`" é uma operação

  Semântica: execute A, então execute B

- If A, B são operações e `T` é um teste, "`if (T) A else B`" é uma operação

  Semântica: se `T` é verdadeiro execute A, senão B

Operações compostas: programas

Programas são construídos recursivamente a partir de operações básica

- Se A, B são operações, então concatenação "`A;B`" é uma operação

  Semântica: execute A, então execute B

- If A, B são operações e T é um teste, "`if (T) A else B`" é uma operação

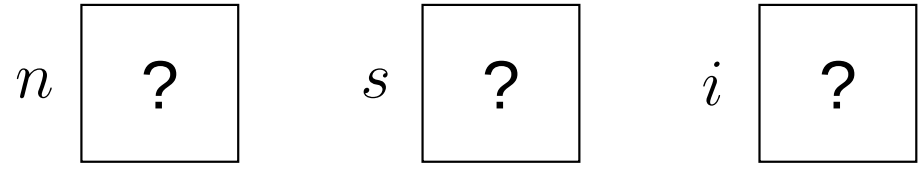
  Semântica: se T é verdadeiro execute A, senão B

- Se A é uma operação e T é um teste, "`while (T) A`" é uma operação

  Semântica: `1:(se (T) A senão (go to 2)) (go to 1) 2:`

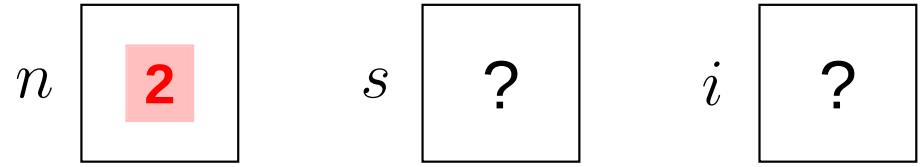
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $\quad s = s + i$;
6: $\quad i = i + 1$;
7: **end while**
8: output s;



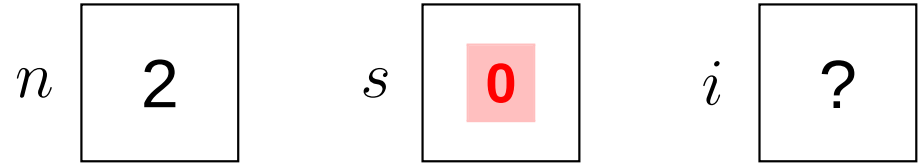
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



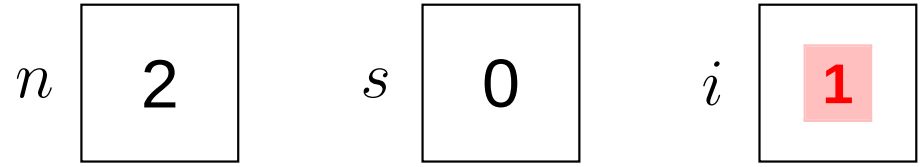
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



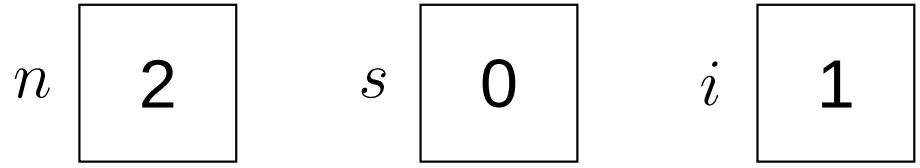
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



Um exemplo

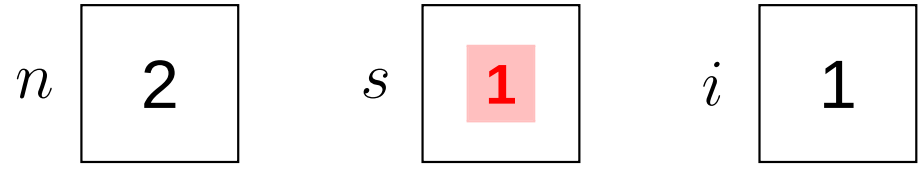
1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



$i \leq n \equiv 1 \leq 2$: true

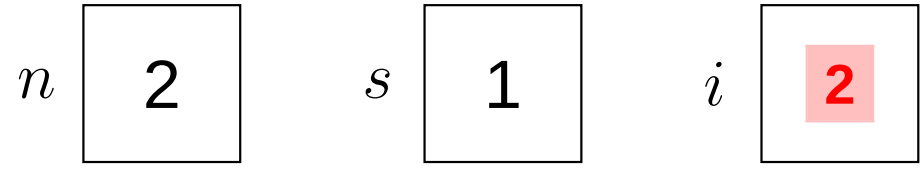
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



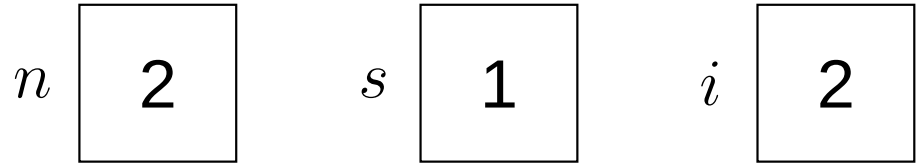
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



Um exemplo

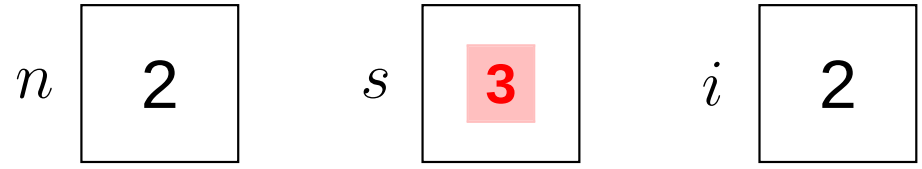
1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



$i \leq n \equiv 2 \leq 2$: true

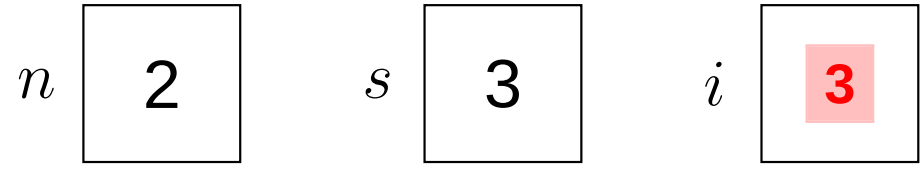
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



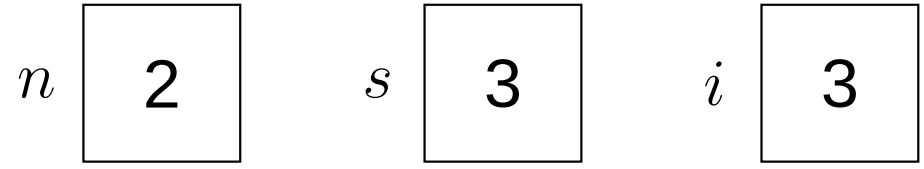
Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



Um exemplo

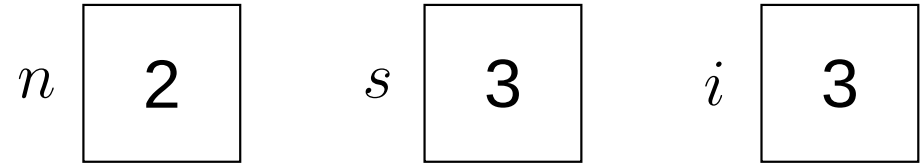
1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



$i \leq n \equiv 3 \leq 2$: false

Um exemplo

1: input n;
2: `int` $s = 0$;
3: `int` $i = 1$;
4: **while** $(i \leq n)$ **do**
5: $s = s + i$;
6: $i = i + 1$;
7: **end while**
8: output s;



output $s = 3$

Complexidade

Complexidade

- Inúmeros programas diferentes podem levar ao mesmo resultado: qual é então o melhor?
- Avalia-se a sua complexidade de tempo (e/ou espaço)
  - **time complexity**: quantas "operações básicas"
  - **space complexity**: quanta memória

  usada pelo programa durante sua execução
- *Pior caso*: valores máximos durante uma execução
- *Melhor caso*: valores mínimo durante uma execução
- *Caso médio*: valores médios

P: um programa
t_P: número de operações básicas realizadas por P

Time complexity (pior caso)

- $\forall P \in \{\texttt{atribuição}, \texttt{aritmética}, \texttt{teste}\}$:

$$\boxed{t_P = 1}$$

Time complexity (pior caso)

- $\forall P \in \{\texttt{atribuição}, \texttt{aritmética}, \texttt{teste}\}$:

$$\boxed{t_P = 1}$$

- **Concatenação**: para P, Q programas:

$$\boxed{t_{P;Q} = t_P + t_Q}$$

Time complexity (pior caso)

- $\forall P \in \{\texttt{atribuição}, \texttt{aritmética}, \texttt{teste}\}$:

$$\boxed{t_P = 1}$$

- **Concatenação**: para P, Q programas:

$$\boxed{t_{P;Q} = t_P + t_Q}$$

- **Teste**: para P, Q programas e R um teste:

$$\boxed{t_{\texttt{if } (T)\ P\ \texttt{else } Q} = t_T + \max(t_P, t_Q)}$$

$\max$: política do pior caso

Time complexity (pior caso)

- $\forall P \in \{\texttt{atribuição}, \texttt{aritmética}, \texttt{teste}\}$:

$$\boxed{t_P = 1}$$

- **Concatenação**: para P, Q programas:

$$\boxed{t_{P;Q} = t_P + t_Q}$$

- **Teste**: para P, Q programas e R um teste:

$$\boxed{t_{\texttt{if } (T)\ P\ \texttt{else } Q} = t_T + \max(t_P, t_Q)}$$

$\max$: política do pior caso

- **Loop**: é um pouquinho mais complicado

(depende em como e quando o loop termina)

Exemplo de complexidade de um loop

O loop completo

Seja P o programa a seguir:

1: $i = 0$;
2: **while** $(i < n)$ **do**
3: A;
4: $i = i + 1$;
5: **end while**

- Assuma que A não muda o valor de i
- Corpo do loop executado n vezes
- $t_P(n) = 1 + n(t_A + 3)$
- *Por que '3'? Bem,* $t_{(i<n)} = 1$, $t_{(i+1)} = 1$, $t_{(i=\cdot)} = 1$

Ordens de complexidade

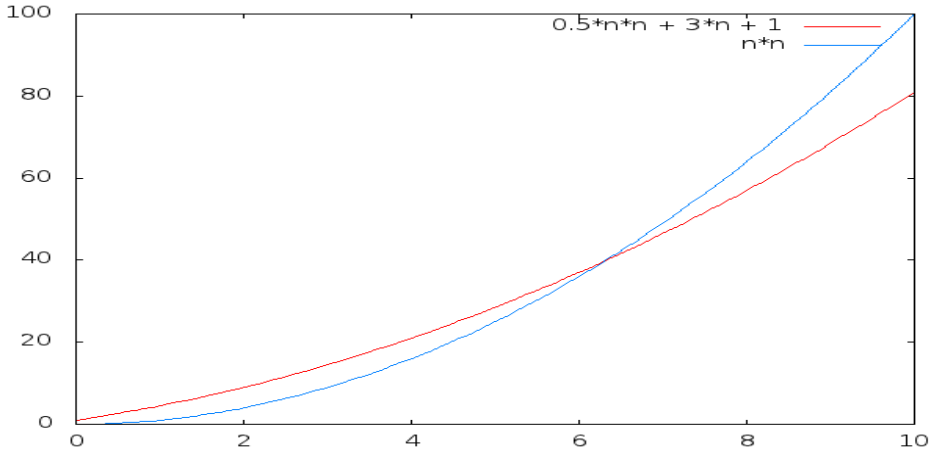
- No programa acima, suponha $t_A = \frac{1}{2}n$

Ordens de complexidade

- No programa acima, suponha $t_A = \frac{1}{2}n$
- Então $t_P = \frac{1}{2}n^2 + 3n + 1$

Ordens de complexidade

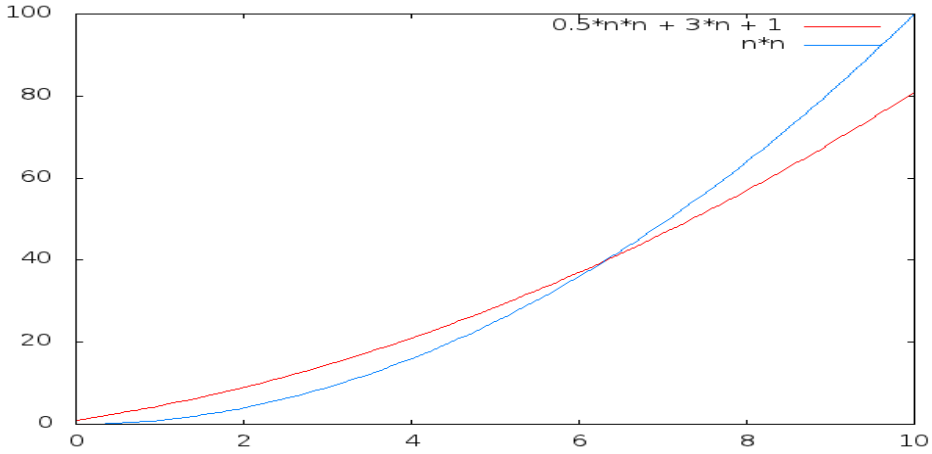
- No programa acima, suponha $t_A = \frac{1}{2}n$
- Então $t_P = \frac{1}{2}n^2 + 3n + 1$
- Ninguém se importa com as constantes $2, 3$: o que importa é que t_P "se comporta melhor do" que a função n^2



$\frac{1}{2}n^2 + 3n + 1$ is $O(n^2)$

Ordens de complexidade

- No programa acima, suponha $t_A = \frac{1}{2}n$
- Então $t_P = \frac{1}{2}n^2 + 3n + 1$
- Ninguém se importa com as constantes $2, 3$: o que importa é que t_P "se comporta melhor do" que a função n^2



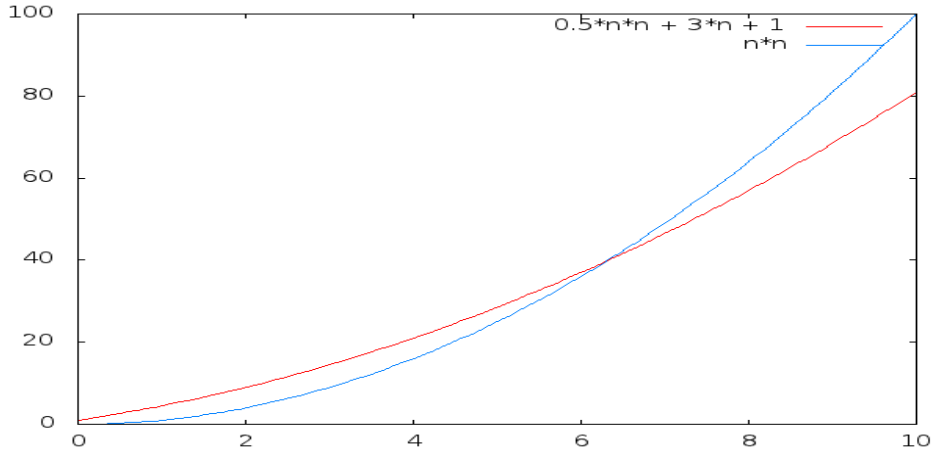
$\frac{1}{2}n^2 + 3n + 1$ is $O(n^2)$

- A função $f(n)$ é *da ordem de* $g(n)$ (notação: $O(g(n))$) se:

$$\exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n > n_0\ (f(n) \leq cg(n)) \qquad (4)$$

Ordens de complexidade

- No programa acima, suponha $t_A = \frac{1}{2}n$
- Então $t_P = \frac{1}{2}n^2 + 3n + 1$
- Ninguém se importa com as constantes $2, 3$: o que importa é que t_P "se comporta melhor do" que a função n^2



$\frac{1}{2}n^2 + 3n + 1$ is $O(n^2)$

- A função $f(n)$ é *da ordem de* $g(n)$ (notação: $O(g(n))$) se:

$$\exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n > n_0\ (f(n) \leq cg(n)) \qquad (5)$$

- Para $\frac{1}{2}n^2 + 3$, $c = 1$ e $n_0 = 2$

Alguns exemplos

Funções	*Ordem*
$an + b$ com a, b constantes	$O(n)$
polinômio de grau d' em n	$O(n^d)$ com $d \geq d'$
$n + \log n$	$O(n)$
$n + \sqrt{n}$	$O(n)$
$\log n + \sqrt{n}$	$O(\sqrt{n})$
$n \log n^3$	$O(n \log n)$
$\frac{an+b}{cn+d}$, a, b, c, d constantes	$O(1)$

- Faça sempre um esforço de achar a melhor função (a que cresce mais devagar) $g(n)$ ao dizer "$f(n)$ é $O(g(n))$"
- Ninguém diz que $2n + 1$ é $O(n^4)$ (embora tecnicamente seja verdade) — diz-se sim que $2n + 1$ é $O(n)$

Observação

- A ordem de complexidade é uma descrição assintótica de $t_P(n)$
- Se $t_P(n)$ não depende de n, sua ordem de complexidade é $O(1)$ (i.e. constante)
- **Exemplo**: looping 10^{1000} vezes um código $O(1)$ ainda resulta em um programa $O(1)$
- Em outras palavras, n precisa ser um parâmetro do programa para que a ordem de complexidade não seja $O(1)$.

Complexidade de loops simples

```
1: input n;
2: int  s = 0;
3: int  i = 1;
4: while (i ≤ n) do
5:     s = s + i;
6:     i = i + 1;
7: end while
8: output s;
```

- $t(n) = 3 + 5n + 1 = 5n + 4$
- $\Rightarrow t(n)$ is $O(n)$

Complexidade de loops simples

```
1: input n;
2: int  s = 0;
3: int  i = 1;
4: while (i ≤ n) do
5:     s = s + i;
6:     i = i + 1;
7: end while
8: output s;
```

- $t(n) = 3 + 5n + 1 = 5n + 4$
- $\Rightarrow t(n)$ is $O(n)$

```
1: for i = 0; i < n − 1; i = i + 1 do
2:     for j = i + 1; j < n; j = j + 1 do
3:         print i, j;
4:     end for
5: end for
```

- $t(n) = 1 + \underbrace{(5(n-1)+6) + \ldots + (5+6)}_{n-1}$
  $= 1 + 5((n-1) + \ldots + 1) + 6(n-1) = \frac{5}{2}n(n-1) + 6n - 5$
  $= \frac{5}{2}n^2 + \frac{7}{2}n - 5$
- $t(n)$ is $O(n^2)$

Arrays

Como vetores na matemática

- Um vetor $x \in \mathbb{Q}^n$ é uma n-tupla $(x_1, \ldots, x_n)$ para $n \in \mathbb{N}$
- Na computação: x é o nome para um endereço de memória com n células sucessivas
- Indexação começa a partir do 0 (última célula é chamada x_{n-1})

x :

x_0	x_1	x_2	x_3	x_4

Como vetores na matemática

- Um vetor $x \in \mathbb{Q}^n$ é uma n-tupla $(x_1, \ldots, x_n)$ para $n \in \mathbb{N}$
- Na computação: x é o nome para um endereço de memória com n células sucessivas
- Indexação começa a partir do 0 (última célula é chamada x_{n-1})

x :

x_0	x_1	x_2	x_3	x_4

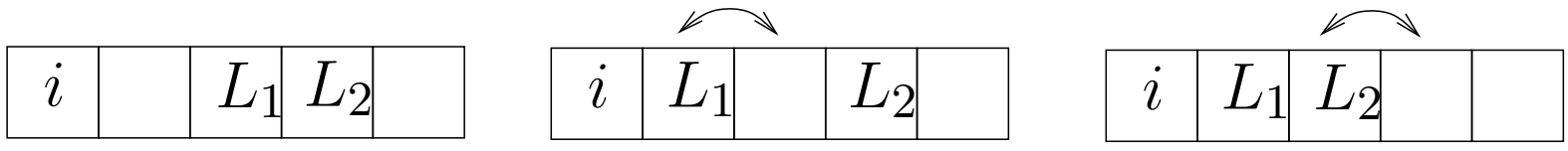
- Um array é **allocado** quando a memória é reservada
- O tamanho do array é decidido no momento de sua alocação
- Normalmente, o temanho do array não muda
- Quando o array não é mais útil, a memória reservada para ele pode ser **desalocada** ou **liberada**

Operações com arrays

Para um array de tamanho n:

Operações	*Complexidade*
Ler valor da i-ésima componente	$O(1)$
Escrever valor na i-ésima componente	$O(1)$
Tamanho	$O(1)$
Remover elemento (célula)	$O(n)$
Inserir elemento (célula)	$O(n)$
Mover subsequência para posição i	$O(n)$

Mover subsequência L para posição i:
extrai subsequência contígua L do array, e a reinsere depois da posição i e antes da posição $i+1$



Norma de um vetor em $\mathbb{R}^5$

1: input $x \in \mathbb{Q}^5$;
2: `int` $i = 0$;
3: `float` $a = 0$;
4: **while** $(i < 5)$ **do**
5: $a = a + x_i \times x_i$;
6: **end while**
7: $a =$ `sqrt`(a);

- Calcula $\sqrt{\sum_{i=0}^{4} x_i^2}$
- Complexidade: $O(1)$ (por quê?)

Loop incompleto

1: input $x \in \{0,1\}^n$;
2: `int` $i = 0$;
3: **while** $(i < n \wedge x_i = 1)$ **do**
4: $x_i = 0$;
5: $i = i + 1$;
6: **end while**
7: **if** $(i < n)$ **then**
8: $x_i = 1$;
9: **end if**
10: output x;

Input	*Output*
(0,0,0,0)	(1,0,0,0)
(1,1,0,0)	(0,0,1,0)
(0,1,1,0)	(1,1,1,0)
(1,1,1,1)	(0,0,0,0)

- Componentes de x podem ser apenas 0 ou 1
- Loop continua sobre todas as componentes enquanto seu valor for 1; na primeira componente 0, o algoritmo para.
- Complexidade?

Complexidade de pior caso do exemplo anterior

- Entre todas as possíveis entradas do algoritmo, encontre aquela que resulta na pior complexidade
- No caso acima, $x = (1, 1, \ldots, 1)$ sempre faz o loop continuar até o fim, i.e. para n iterações

Thm.

$(1, 1 \ldots, 1)$ é a entrada que resulta na pior complexidade

Proof

Suponha que isto seja falso, então existe um vetor $x \neq (1, \ldots, 1)$ resultando em $t(n) > n$. Uma vez que $x \neq (1, \ldots, 1)$, x contem pelo menos um componente 0. Seja $j < n$ o menor índice de um componente tal que $x_j = 0$: na iteração j o loop termina, e $t(n) = j$, que é menor do que n: contradição.

- Dado que as outras operações são $O(1)$, temos $O(n)$

Complexidade de pior caso do exemplo anterior

- Entre todas as possíveis entradas do algoritmo, encontre aquela que resulta na pior complexidade
- No caso acima, $x = (1, 1, \ldots, 1)$ sempre faz o loop continuar até o fim, i.e. para n iterações

Thm.

$(1, 1 \ldots, 1)$ é a entrada que resulta na pior complexidade

Proof

Suponha que isto seja falso, então existe um vetor $x \neq (1, \ldots, 1)$ resultando em $t(n) > n$. Uma vez que $x \neq (1, \ldots, 1)$, x contem pelo menos um componente 0. Seja $j < n$ o menor índice de um componente tal que $x_j = 0$: na iteração j o loop termina, e $t(n) = j$, que é menor do que n: contradição.

- Dado que as outras operações são $O(1)$, temos $O(n)$

Dificuldade desta abordagem: identificar a "pior" entrada e provar que ela é de fato a pior dentre todas as outras.

Complexidade de caso médio do exemplo anterior (1/2)

- A análise de caso médio necessita de um espaço de probabilidades:
  - assuma que o evento $x_i = b$ é independente dos eventos $x_j = b$ para todo $i \neq j$
  - assuma que cada célula x_i do array contem 0 ou 1 com igual probabilidade $\frac{1}{2}$

Complexidade de caso médio do exemplo anterior (1/2)

- A análise de caso médio necessita de um espaço de probabilidades:
  - assuma que o evento $x_i = b$ é independente dos eventos $x_j = b$ para todo $i \neq j$
  - assuma que cada célula x_i do array contem 0 ou 1 com igual probabilidade $\frac{1}{2}$
- Para qualquer vetor tendo as primeiras $k+1$ componentes $(\underbrace{1, \ldots, 1}_{k}, 0)$, o loop é executado k vezes

  (para todo $0 \leq k < n$)

Vetor binário tendo as primeiras k componentes iguais a 1 tem probabilidade $\left(\frac{1}{2}\right)^{k+1}$

Complexidade de caso médio do exemplo anterior (1/2)

- A análise de caso médio necessita de um espaço de probabilidades:
  - assuma que o evento $x_i = b$ é independente dos eventos $x_j = b$ para todo $i \neq j$
  - assuma que cada célula x_i do array contem 0 ou 1 com igual probabilidade $\frac{1}{2}$
- Para qualquer vetor tendo as primeiras $k+1$ componentes $(\underbrace{1, \ldots, 1}_{k}, 0)$, o loop é executado k vezes

  (para todo $0 \leq k < n$)

  Vetor binário tendo as primeiras k componentes iguais a 1 tem probabilidade $\left(\frac{1}{2}\right)^{k+1}$
- Se o vetor é $(\underbrace{1, \ldots, 1}_{n})$ o loop é executado n vezes

  A probabilidade de termos este vetor é $\left(\frac{1}{2}\right)^{n}$

Complexidade de caso médio do exemplo anterior (2/2)

- O loop é executado k vezes com probabilidade $\left(\frac{1}{2}\right)^{k+1}$, para $k < n$

Complexidade de caso médio do exemplo anterior (2/2)

- O loop é executado k vezes com probabilidade $\left(\frac{1}{2}\right)^{k+1}$, para $k < n$
- O loop é executado n vezes com probabilidade $\left(\frac{1}{2}\right)^{n}$

Complexidade de caso médio do exemplo anterior (2/2)

- O loop é executado k vezes com probabilidade $\left(\frac{1}{2}\right)^{k+1}$, para $k < n$
- O loop é executado n vezes com probabilidade $\left(\frac{1}{2}\right)^{n}$
- Valor esperado para o número de iterações do loop:

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n} \leq \sum_{k=0}^{n-1} k2^{-k} + n2^{-n} = \sum_{k=0}^{n} k2^{-k}$$

Complexidade de caso médio do exemplo anterior (2/2)

- O loop é executado k vezes com probabilidade $\left(\frac{1}{2}\right)^{k+1}$, para $k < n$
- O loop é executado n vezes com probabilidade $\left(\frac{1}{2}\right)^{n}$
- Valor esperado para o número de iterações do loop:

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n} \leq \sum_{k=0}^{n-1} k2^{-k} + n2^{-n} = \sum_{k=0}^{n} k2^{-k}$$

Thm.

$\lim_{n\to\infty} \sum_{k=0}^{n} k2^{-k} = 2$

Proof

Série geométrica $\sum_{k\geq 0} q^k = \frac{1}{1-q}$ for $q \in [0,1)$. Derivando w.r.t. q, obtem-se $\sum_{k\geq 0} kq^{k-1} = \frac{1}{(1-q)^2}$; multiplicando-se por q, tem-se $\sum_{k\geq 0} kq^k = \frac{q}{(1-q)^2}$. Para $q = \frac{1}{2}$, tem-se $\sum_{k\geq 0} k2^{-k} = (1/2)/(1/4) = 2$.

Complexidade de caso médio do exemplo anterior (2/2)

- O loop é executado k vezes com probabilidade $\left(\frac{1}{2}\right)^{k+1}$, para $k < n$
- O loop é executado n vezes com probabilidade $\left(\frac{1}{2}\right)^{n}$
- Valor esperado para o número de iterações do loop:

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n} \leq \sum_{k=0}^{n-1} k2^{-k} + n2^{-n} = \sum_{k=0}^{n} k2^{-k}$$

Thm.

$\lim_{n\to\infty} \sum_{k=0}^{n} k2^{-k} = 2$

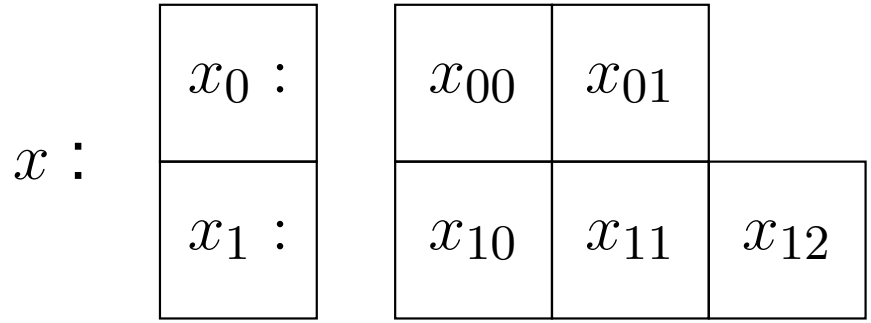
Proof

Série geométrica $\sum_{k\geq 0} q^k = \frac{1}{1-q}$ for $q \in [0,1)$. Derivando w.r.t. q, obtem-se $\sum_{k\geq 0} kq^{k-1} = \frac{1}{(1-q)^2}$; multiplicando-se por q, tem-se $\sum_{k\geq 0} kq^k = \frac{q}{(1-q)^2}$. Para $q = \frac{1}{2}$, tem-se $\sum_{k\geq 0} k2^{-k} = (1/2)/(1/4) = 2$.

Portanto, a complexidade de caso médio é constante $O(1)$

Arrays “denteados”

- **Array denteado**: um vetor cujas componentes são vetores de tamanhos diferentes
- E.g. $x = ((x_{00}, x_{01}), (x_{10}, x_{11}, x_{12}))$



Arrays “denteados”

- **Array denteado**: um vetor cujas componentes são vetores de tamanhos diferentes
- E.g. $x = ((x_{00}, x_{01}), (x_{10}, x_{11}, x_{12}))$

x :

x_0 :	x_{00}	x_{01}	
x_1 :	x_{10}	x_{11}	x_{12}

- **Caso especial**: quantos todos os subvetores têm o mesmo tamanho, temos uma matriz: `int x[][] = new int [2][3];`

$$x = \begin{pmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \end{pmatrix}$$

Representando relações

- Arrays denteados podem ser usados para representar relações em um conjunto finito

Representando relações

- Arrays denteados podem ser usados para representar relações em um conjunto finito
- Seja $V = \{v_1 \ldots, v_n\}$ e E um conjunto de relações entre elementos de V

  E é um conjunto de pares ordenados (u, v)

Representando relações

- Arrays denteados podem ser usados para representar relações em um conjunto finito
- Seja $V = \{v_1 \ldots, v_n\}$ e E um conjunto de relações entre elementos de V

  E é um conjunto de pares ordenados (u, v)

- **Representação**:
  - array de n componentes
  - a i-ésima componente é o array das componentes v_j relacionadas a v_i

Representando relações

- Arrays denteados podem ser usados para representar relações em um conjunto finito
- Seja $V = \{v_1 \ldots, v_n\}$ e E um conjunto de relações entre elementos de V

  E é um conjunto de pares ordenados (u, v)

- **Representação**:
  - array de n componentes
  - a i-ésima componente é o array das componentes v_j relacionadas a v_i
- Exemplo: $V = \{1, 2, 3\}$,
  $E = \{(1,1), (1,2), (2,3), (3,1), (3,2), (3,3)\}$

E :

1	1	2	
2	3		
3	1	2	3

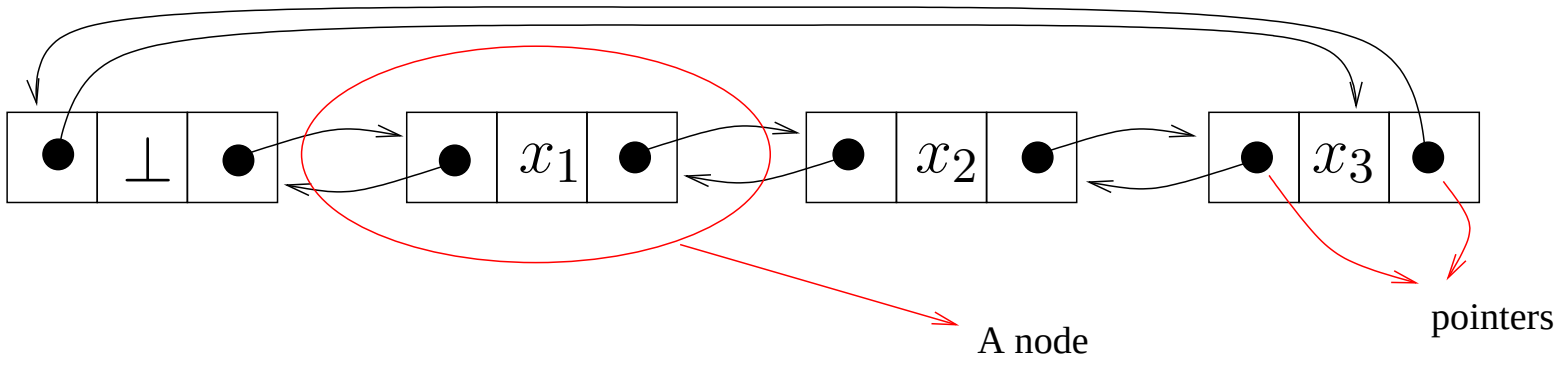
Applicação: Redes

Deficiências no uso de arrays

- “Essencialmente” de tamanho fixo
- Tamanho precisa ser conhecido a priori
- A mudança de posições relativas de elementos é ineficiente

Listas

Lista duplamente encadeada



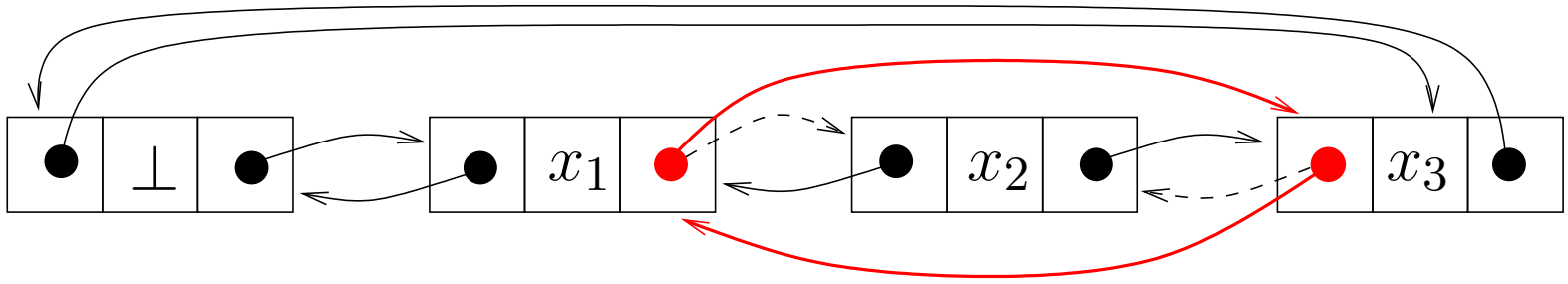
- **Né** N: um elemento da lista

N.prev	$=$	endereço do nó anterior na lista
N.next	$=$	endereço do nó posterior na lista
N.datum	$=$	dado armazenado no nó

- **nós sentinela** $\perp$: antes do primeiro elemento, depois do último elemento, nenhum dado armazenado
- *Todo nó tem dois ponteiros, e é apontado por dois outros nós*

Remova um nó

Remove nó atual (`this`)



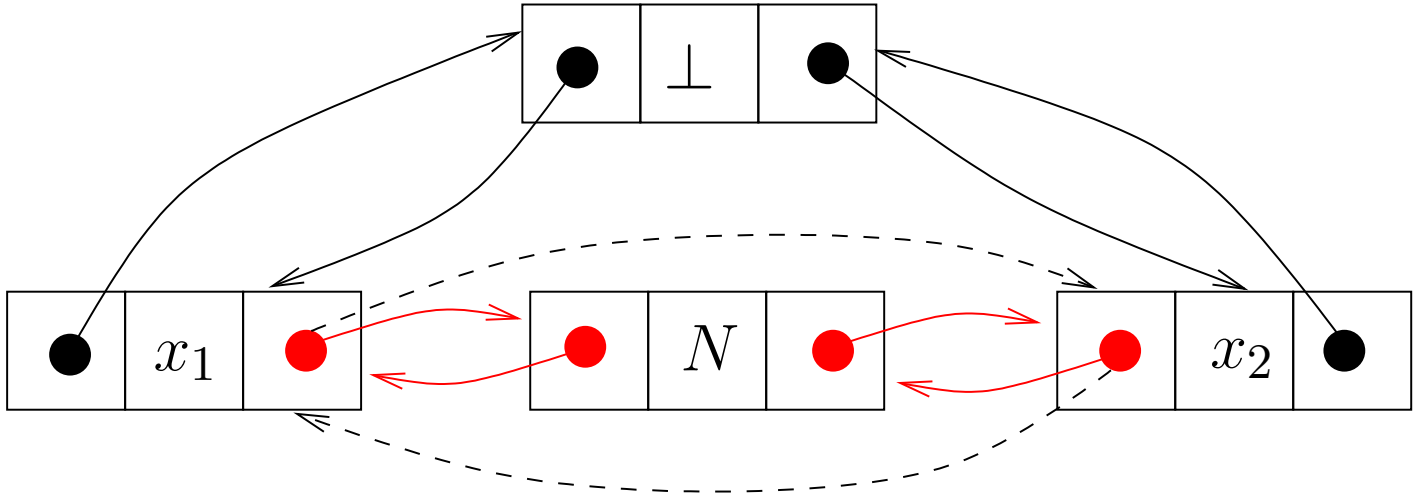
Neste exemplo, `this`$= X_2$

```
1: this.prev.next = this.next ;
2: this.next.prev = this.prev ;
```

Complexidade de pior caso: $O(1)$

Insere um nó

Insere nó atual (`this`) após nó X_1



Neste exemplo, `this`$= N$

1: `this.prev` $= X_1$;
2: `this.next` $= X_1$`.next` ;
3: X_1`.next` $=$ `this` ;
4: `this.next.prev` $=$ `this` ;

Complexidade de pior caso: $O(1)$

Find next

- Dada uma lista L e um nó X, encontre a próxima ocorrência do elemento b
- Se $b \in L$ retorne o nó onde b é armazenado, senão retorne $\perp$

1: **while** $(X.\texttt{datum} \neq b \wedge X \neq \perp)$ **do**
2: $\quad X = X.\texttt{next}$
3: **end while**
4: **return** X

Warning: *todo teste custa 2 operações básicas, ineficiente*

Find next

- Dada uma lista L e um nó X, encontre a próxima ocorrência do elemento b
- Se $b \in L$ retorne o nó onde b é armazenado, senão retorne $\perp$

1: **while** $(X.\texttt{datum} \neq b \wedge X \neq \perp)$ **do**
2: $\quad X = X.\texttt{next}$
3: **end while**
4: **return** X

Warning: *todo teste custa 2 operações básicas, ineficiente*

1: $\perp.\texttt{datum} = b$
2: **while** $(X.\texttt{datum} \neq b)$ **do**
3: $\quad X = X.\texttt{next}$
4: **end while**
5: **return** X

Agora $t_{\text{test}} = 1$

Operações em listas

Para uma lista duplamente encadeada de tamanho n:

Operação	*Complexidade*
Read/write valor do i-ésimo nó	$O(n)$
Size	$O(1)$
Find next	$O(n)$
Está vazia?	$O(1)$
Read/write valor do primeiro/último nó	$O(1)$
Remover elemento	$O(1)$
Inserir elemento	$O(1)$
Mover subsequência para posição i	$O(1)$
Concatenar	$O(1)$

Fim do módulo 1